FINDS: An N-Body Simulator for Large Schools of Fish

Mufaro Machaya[1] *, Mohamed Niged Mabrouk[2] †, and Daniel Floryan[2] ‡

August 2026

[1]Department of Physics and Astronomy, University of British Columbia
[2]Department of Mechanical and Aerospace Engineering, University of Houston

Abstract

Animals traveling in groups yield many benefits, such as security against predators and energy efficiency, and understanding this behavior could have key applications to fields like robotics. To model all realistic schools of fish, computational models must accommodate hundreds of millions of fish, but current simulations are limited to very small group sizes because the passive hydrodynamic model of fish is an N-body problem. Thus, FINDS, or the Fish INteraction and Dynamics Simulator, was developed from the source/sink-dipole model developed by Mabrouk and Floryan to streamline simulations using standard N-body algorithms. First, FINDS contains routines for computing the time-derivative of a school through brute force computation, the Barnes-Hut approximation, and the fast multipole method; several adaptive and non-adaptive Runge-Kutta numerical integration methods from first to sixth order; a small library of post-processing modules for reading, writing, and analyzing the simulation datasets; and simple yet powerful interfaces for defining precise initial conditions for a wide variety of simulations using direct calculations or configuration files. FINDS is validated graphically and numerically against past simulations performed by Mabrouk and Floryan, proving that it is computationally equivalent (within the expected error of the Barnes-Hut and fast multipole methods) while being considerably more efficient.

1 Introduction

Synchronous mass movement is a common pattern throughout nature, observed most commonly in the movement of smaller animals or prey animals such as birds, fish, bugs, oxen, deer, etc. due to the multitude of benefits that members of such groups gain [1, 2]. As summarized by Mabrouk and Floryan 2024 and 2025, members of schools of fish gain major advantages in individual fitness due to various effects resulting from existing in a group rather than as an individual, including (but not limited to) stronger shielding from predators, improved navigation, eased access to reproduction, and most importantly, a greater energy efficiency during travel [3, 4]. As such, scientists and engineers developing analogous technologies like drones are keenly interested in understanding this behavior so that it can be replicated in swarming robots for purposes such as agriculture, infrastructure development and maintenance, and military operations [5].

In the case of schools of fish, prior models have largely focused on behavioral causes of group cohesion, but these models typically fail to explain the effect of passive hydrodynamics in maintaining the flow of these systems [6, 7]. For this reason, models of swimmers as dipoles in free space have been developed to model this passive hydrodynamic interaction, such as Mabrouk and Floryan's source/sink-dipole Tchieu et al.'s vortex-dipole models [3, 4, 8].

However, the larger issue with modeling groups of interacting swimmers is computational rather than theoretical, because models containing non-hierarchical interactions where every member of the system affects every other member (i.e., models that are not simply "following the leader" with a central member or a set of central members that determine the entirety of the group's motion) form an N-body problem. In particular, dipole models compute interactions from each of the poles of each swimmer to each of the poles of each other swimmer individually, and thus, a simulation of N swimmers becomes analogous to a particle simulation problem for $2N$ particles with $4N^2$ interactions on every differential time-step. This severely limits the size of schools that can be studied computationally, and as such, most if not all of the past research utilizing dipole models of swimmers has focused on systems at or below 100,000 swimmers.

Given that real schools of fish have been estimated to contain upwards of 250 million swimmers, an approximation method to reduce the computational complexity of this simulation from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ or $\mathcal{O}(N)$ is necessary to explore group cohesion due to passive hydrodynamics in massive systems [9]. Luckily, the problem of reducing the computational complexity of N-body particle simulations has been thoroughly studied across the fields of molecular dynamics, computational electromagnetics, gravitational astrophysics, and more [10], and this has yielded algorithms such as the Barnes-Hut algorithm and the fast multipole method that simplify calculations by distinguishing between close and

*Corresponding author: mufaro2@student.ubc.ca
†mmabrouk@uh.edu
‡dfloryan@uh.edu

far particles and using clustered approximations for far-field interactions [10, 11, 12].

2 Mathematical Model of Swimmers

FINDS is developed around the Mabrouk and Floryan's source/sink dipole model for far-field interactions [3, 4]. Under this model, each fish i is defined as a dipole consisting of a fluid source at the head of the fish $\hat{\mathbf{x}}_{f,i}$, and a fluid sink at the tail $\hat{\mathbf{x}}_{b,i}$, separated by the length of the fish ℓ_i. The source and the sink of each fish are defined to have the same volumetric flow rate, σ_i, to ensure the conservation of fluid by making the sinks consume exactly the same amount of fluid as the sources. For simplicity, the source and sink (or "front" and "back") of each fish are called its "features."

Each fish is defined as an eight-dimensional vector consisting of four variables: $\vec{\mathbf{x}}_{c,i}$, the three-dimensional center-of-mass position vector, $\hat{\mathbf{n}}_i$, the three-dimensional orientation unit vector, ℓ_i, and σ_i. Together, the collection of each of these fish vectors at a given time forms the state vector $\vec{\mathbf{X}}$.

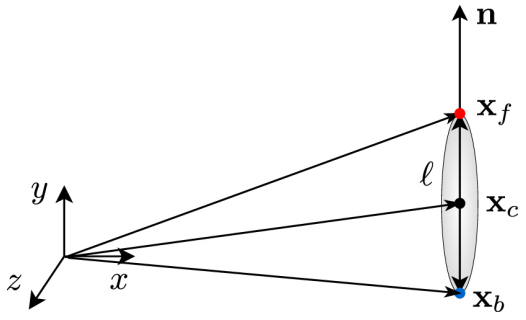

Figure 1: *Visualization of the model for a given swimmer. Taken from Mabrouk and Floryan 2024 [3].*

In particular, $\vec{\mathbf{x}}_{c,i}$ and $\hat{\mathbf{n}}_i$ are dynamic and ℓ_i and σ_i are constant for each fish i. This formalism leaves the translational and rotational dynamics for each fish as the system of two coupled first-order differential equations concerning $\vec{\mathbf{x}}_{c,i}$ and $\hat{\mathbf{n}}_i$ respectively. With this, the derivatives of each are defined as

$$\frac{\mathrm{d}\vec{\mathbf{x}}_{c,i}}{\mathrm{d}t} = \frac{1}{2}\left(\frac{\mathrm{d}\vec{\mathbf{x}}_{f,i}}{\mathrm{d}t} + \frac{\mathrm{d}\vec{\mathbf{x}}_{b,i}}{\mathrm{d}t}\right) \tag{1}$$

$$\frac{\mathrm{d}\hat{\mathbf{n}}_i}{\mathrm{d}t} = \frac{\Delta\vec{\mathbf{v}}_i + 2\lambda_i\hat{\mathbf{n}}_i}{\ell_i} \tag{2}$$

where

$$\lambda_i = \frac{\Delta\vec{\mathbf{v}}_i \cdot \hat{\mathbf{n}}_i}{2} \tag{3}$$

is a Lagrange multiplier to ensure that ℓ_i is kept constant, and

$$\Delta\vec{\mathbf{v}}_i = \frac{\mathrm{d}\vec{\mathbf{x}}_{f,i}}{\mathrm{d}t} - \frac{\mathrm{d}\vec{\mathbf{x}}_{b,i}}{\mathrm{d}t}. \tag{4}$$

With this, the translational and rotational dynamics for each fish are variably dependent on the translational derivatives of the source and sink for each fish, and this reduces the problem of modeling fish to an analogous problem of simple Laplacian particle dynamics based on treating the sources and sinks as individual particles with charge σ_i. Under this lens, the potential function resulting from a particle is the charge multiplied by Green's function for the three-dimensional Laplace equation,

$$\Phi_i(r) = \sigma_i G(r) = -\frac{\sigma_i}{4\pi r}. \tag{5}$$

However, as this model is designed for far-field interactions, it becomes unphysical for small distances, as infinitely small distances produce infinitely large potential. To solve this problem, Mabrouk and Floryan reformulated this potential with a regularization parameter ε[1], defined as

$$\Phi_{i,\varepsilon}(r) = \Phi_i(r)\,\mathrm{erf}(r/\varepsilon). \tag{6}$$

From the potential functions, the velocity field resulting from any source or sink is defined as the gradient of the potential function

$$\vec{\mathbf{u}}_i(\vec{\mathbf{r}}) = \boldsymbol{\nabla}\Phi_i = \frac{\sigma_i}{4\pi}\frac{\vec{\mathbf{r}}}{r^3} \tag{7}$$

$$\vec{\mathbf{u}}_{i,\varepsilon}(\vec{\mathbf{r}}) = \boldsymbol{\nabla}\Phi_{i,\varepsilon} = \vec{\mathbf{u}}_i(\vec{\mathbf{r}})\xi_\varepsilon(r) \tag{8}$$

where $\xi_\varepsilon(r)$ is the regularization coefficient

$$\xi_\varepsilon(r) = \mathrm{erf}(r/\epsilon) - \frac{2r}{\varepsilon\sqrt{\pi}}\exp\left(-\frac{r^2}{\varepsilon^2}\right). \tag{9}$$

With the velocity fields well-defined, the velocity of a given source or sink α can be defined generally as

$$\frac{\mathrm{d}\vec{\mathbf{x}}_{\alpha,i}}{\mathrm{d}t} = U_i\hat{\mathbf{n}}_i + \sum_{i\neq j}^{N}\left[\vec{\mathbf{u}}_j(\vec{\mathbf{r}}_{\alpha f}) - \vec{\mathbf{u}}_j(\vec{\mathbf{r}}_{\alpha b})\right]. \tag{10}$$

where the first term is the *internal* velocity contribution and the second term is the *external* velocity contribution. In detail,

$$U_i = \frac{\sigma_i}{4\pi\ell_i^2} \tag{11}$$

is the self-propelled speed resulting from the interaction between any source-sink pair of a dipole, and the second term represents the sum of the velocity contribution to a source or sink from a dipole, where

$$\vec{\mathbf{r}}_{\alpha\beta} = \vec{\mathbf{x}}_{\alpha,i} - \vec{\mathbf{x}}_{\beta,j} \tag{12}$$

is the displacement between feature α of fish i and feature β of fish j. In the case of regularization, $\vec{\mathbf{u}}_j$ is simply replaced with $\vec{\mathbf{u}}_{j,\varepsilon}$.

[1] Mabrouk and Floryan, personal communication, July 2026.

3 Derivative Computation

As stated previously, the modelling of fish is an N-body problem where the algorithmic complexity of computing a derivative is, through Brute-Force calculations, of order $\mathcal{O}(N^2)$. To reduce this computation time, two tree-based approximation algorithms are implemented: the Barnes-Hut algorithm and the Fast Multipole Method, of orders $\mathcal{O}(N \log N)$ and $\mathcal{O}(N)$ respectively [11, 12].

Brute force computation and the Barnes-Hut approximation are implemented directly into the FINDS code-base, and as such, the user has the ability to switch between unregularized and regularized calculations with ease. However, the Fast-Multipole Method is computed externally using Greengard et al.'s Fast Multipole Method library FMM3D, which will only compute the standard Laplacian gradient $\vec{\mathbf{u}}_i$, so the Fast-Multipole Method cannot be used with regularization [13]. Additionally, all three methods are optionally CPU-parallelized using OpenMP [14].

Method	**Complexity**	**Parameter**
Brute Force	$\mathcal{O}(N^2)$	None
Barnes-Hut	$\mathcal{O}(N \log N)$	θ
FMM	$\mathcal{O}(N)$	ϵ

Table 1: *A table summarizing each of the derivative computation methods. θ is similarly referred to as the "approximation ratio" and ϵ is the numerical precision, and both parameters determine the precision with which each derivative is calculated (with lower values producing more precise measurements).*

3.1 The Barnes-Hut Approximation

As described by Barnes and Hut, the Barnes-Hut algorithm computes 3-dimensional pairwise interactions in $\mathcal{O}(N \log N)$ time by constructing a space-partitioning octree that clusters swimmers that are sufficiently far away from each test swimmer [11].

3.1.1 Linear Octree Construction

FINDS implements a linear octree, or aligned arrays for each node parameter where each array index corresponds to a node in the tree. In particular, each node stores information on both the node itself and the data stored within this node [15, 16]. With regard to the node overall, it stores its center position, $\vec{\mathbf{x}}_i$, the side length of the node, s_i, and indices to its child nodes. With regard to the data stored within this node, the node stores the number of swimmers that are being averaged out in this node n_i, the total volumetric flow rate,

$$\zeta_j = \sum_{j=0}^{n_i} \sigma_j \tag{13}$$

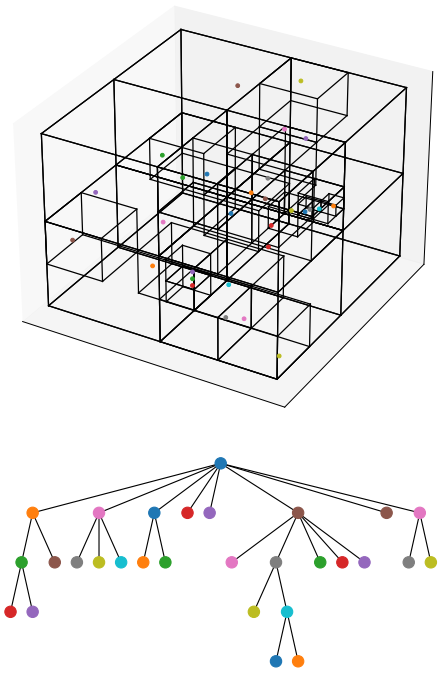

Figure 2: *Top: An example octree constructed in three-dimensional space to partition twenty points, constructed with an arbitrary insertion order. Bottom: The same octree, represented two-dimensionally.*

and weighted sums for the position, orientation, and length. Each sum is weighted by the volumetric flow rate (which can be analogously thought of as the charge or mass in Coulombic or gravitational N-body dynamics), so that the average position, orientation, and length is produced by dividing by the total volumetric flow rate,

$$\overline{\vec{\mathbf{x}}_{c,i}} = \zeta_j^{-1} \sum_{j=0}^{n_i} \vec{\mathbf{x}}_{c,j} \sigma_j \tag{14}$$

$$\overline{\hat{\mathbf{n}}_i} = \zeta_j^{-1} \sum_{j=0}^{n_i} \hat{\mathbf{n}}_j \sigma_j \tag{15}$$

$$\overline{\ell_i} = \zeta_j^{-1} \sum_{j=0}^{n_i} \ell_j \sigma_j \tag{16}$$

where i in this context refers to the node index within the octree lists and j refers to the index of each swimmer stored within this node. This architecture was chosen due to its traversal efficiency when sorted according to a space-filling curve, as this would keep physically close nodes close together in memory, which maximizes cache locality [15, 16].

Construction of the Linear Octree begins by taking a system of swimmers $\vec{\mathbf{X}}$ and computing vectors containing minimum and maximum values along each dimension for the

system, or

$$\vec{\mathbf{x}}_{min} = \langle \min \mathbf{x}, \min \mathbf{y}, \min \mathbf{z} \rangle \tag{17}$$

$$\vec{\mathbf{x}}_{max} = \langle \max \mathbf{x}, \max \mathbf{y}, \max \mathbf{z} \rangle \tag{18}$$

where $\mathbf{x}$ contains all of the x-values for the position of every swimmer in the system, $\mathbf{y}$, the y-values, and $\mathbf{z}$, the z-values. Then, the center position of the root is defined as

$$\vec{\mathbf{x}}_0 = \frac{\vec{\mathbf{x}}_{min} + \vec{\mathbf{x}}_{max}}{2} \tag{19}$$

and the side length for the root node is computed as the maximum component of their difference, or

$$s_0 = \max(\vec{\mathbf{x}}_{max} - \vec{\mathbf{x}}_{min}). \tag{20}$$

Next, each of the swimmers in the system need to be mapped to a space-filling curve to know what order to insert the nodes such that physically close swimmers are inserted in order. To accomplish this task, the 3-D position of each swimmer is mapped to a Morton curve localized to a relative origin point for the tree chosen as minimum position for the root node's bounding cube, or

$$\vec{\mathbf{x}}_{\mathcal{O}} = \vec{\mathbf{x}}_0 - \frac{s_0}{2}\vec{\mathbf{1}}_3. \tag{21}$$

This is done to ensure that the position of every swimmer can be described using only positive numbers. Then, the position of each swimmer is truncated to an integer and a Morton number corresponding to this position is produced by interleaving the binary coordinate values using bit-shifting operations. Note that this operation implicitly assumes that the three numbers going in are each a maximum of 21 bits such that the output Morton number is 64-bit, and this integer-sizing has significant consequences (see section 11).

Then, a position-local array of swimmer indices is produced by first building an in-order list of indices $(0, 1, \ldots, N)$ corresponding to the indices for each swimmer in $\vec{\mathbf{X}}$, then arranging it according to the quick-sorted order of the corresponding Morton numbers for each swimmer position. Then, the linear octree is constructed recursively by defining each node region with the center position $\vec{\mathbf{x}}_j$, side length s_j, tree depth d_j, and the corresponding index range of the swimmer array encompassing all of the swimmers that fall within this node. For example, for the root node, the center position is $\vec{\mathbf{x}}_0$, the side length is s_0, and the range of enclosed swimmers is $[0, N]$. Then, for each swimmer contained within this range, the center-of-mass position $\vec{\mathbf{x}}_{c,j}$, orientation $\hat{\mathbf{n}}_j$, length ℓ_j, and volumetric flow rate σ_j are added accordingly to the corresponding weighted sums as stated previously, and then the weighted average source and sink positions are computed for this node based on the weighted average position, orientation, and length.

Next, the recursion is continued to child nodes, if any. If the range of swimmers that falls within this node is one or no swimmers, then the recursion ends here. If there are two or more swimmers within this octant, then this node cannot be a leaf, and this node is then subdivided into eight child octants. Each child octant is denoted by an index o from 0 to 7, and for each child octant, the range of node indices which should fall within the child octant is computed based on performing bit-shift operations on the associated Morton indices. This range is half-open, so if the start and end are equal, then no swimmers are located within the child node, and a new child node will only be added to the tree recursively if there are any swimmers located within the node. Otherwise, the child node is skipped.

In the case that a new child node is added, its side length $s_{j,o}$ is the side length of the current node divided by two,

$$s_{j,o} = s_j/2, \tag{22}$$

and its center position is computed based on its index, o, as

$$\vec{\mathbf{x}}_{j,o} = \vec{\mathbf{x}}_j + \frac{s_j}{4}\begin{pmatrix} f(o \,\&\, 1) \\ f(o \,\&\, 2) \\ f(o \,\&\, 4) \end{pmatrix} \tag{23}$$

where

$$f(x) = \begin{cases} 1 & \text{if } x = 1 \\ -1 & \text{if } x = 0 \end{cases}. \tag{24}$$

The binary representation of o can also be interpreted as a relative ordering on each axis between the center position of the parent node and the child octant. In order, the bits correspond with the z, y, and x-axes respectively, and 1 and 0 correspond with the center position of the child octant along a given dimension being greater than and less than or equal to that of the parent node for that dimension. Then, for each dimension, if the child position is less than or equal to the parent, then the child's center position is offset by $-s_j/4$. Otherwise, the child's center position is offset by $s_j/4$ along that axis. For example, the index $o = 5$ has binary representation 101, and this corresponds with $+s_j/4$ on the z-axis, $-s_j/4$ on the y-axis, and $+s_j/4$ on the x-axis.

3.1.2 Barnes-Hut Velocity Computation

Once constructed, the velocity contribution is computed in $\mathcal{O}(N \log N)$ time by performing a depth-first tree traversal on the linear octree. We can notate the test swimmer to compute the velocity for as a, the tree index for a given node as i, the weighted average (or composite) swimmer for index i as b_i, and the Barnes-Hut minimum approximation ratio supplied by the user as θ.

Beginning from the root node, if the current node is a leaf ($n_i = 1$) and the position for swimmer b_i is not approximately

equal to the position of swimmer a (within a precision of 10^{-15} to account for floating-point errors) then the velocity contributions are computed immediately via equation 10, and the recursion is ended.

Otherwise, the size-to-distance ratio ϕ_i is computed for this node as

$$\phi_i = \frac{s_i}{d_{ab_i}} \tag{25}$$

where d is the distance from swimmer a to average swimmer b_i,

$$d_{ab_i} = \left\| \vec{\mathbf{x}}_{c,a} - \overline{\vec{\mathbf{x}}_{c,b_i}} \right\|. \tag{26}$$

Then, if $\phi_i < \theta$, then the composite swimmer b_i is said to be *close* to swimmer a, and the velocity contributions are computed immediately via equation 10 and the recursion ends [11]. If $\phi_i \geq \theta$, then composite swimmer b_i is said to be *far* from swimmer a, and we recurse into each of the child nodes of node i [11].

3.2 The Fast Multipole Method (FMM)

As described by Greengard and Rokhlin, the fast multipole method similarly uses tree construction to compute the velocity field as the gradient of the potential field approximated via a multipole expansion of order $p(\epsilon)$ where ϵ (not to be confused with ε defined above) is the user-defined floating-point precision for the calculation [12]. As stated previously, the implementation of the fast multipole method is not a part of FINDS, but rather, the library FMM3D by Greengard et al. is used to compute the Laplacian gradient,

$$\vec{\mathbf{u}}(\vec{\mathbf{x}}) \approx \boldsymbol{\nabla} \left(\sum_{j=1}^{N} \frac{\sigma_j}{4\pi \|\vec{\mathbf{x}} - \vec{\mathbf{x}}_j\|} \right) \tag{27}$$

for each source/sink using the function `lfmm3d_t_c_g` [13]. In essence, this computation is based on treating the source and sink for each dipole as independent particles rather than members of a dipole, so to compute only the external interaction, each source/sink has to have its velocity corrected for double-counting the interaction with its pair. However, this is negligible compared to the overall calculation, as this is computed in $\mathcal{O}(N)$ time overall, so the overall runtime is still $\mathcal{O}(N)$.

3.3 Derivative Calculation Performance Evaluation

As shown through table 1, the three derivative calculation methods employed in FINDS each have a distinct runtime complexity as a function of N, and therefore, have specific ranges of N for which they are the most effective due to their individual scaling orders and proportionality constants.

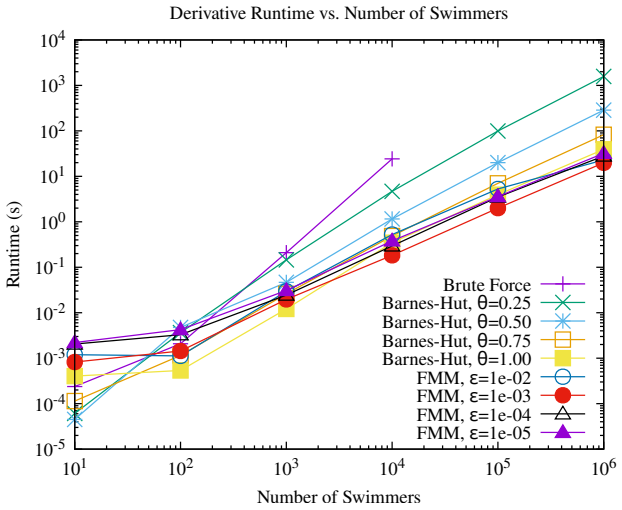

Figure 3: *A logarithmic-scale runtime comparison plot for each derivative computation method when parallelized using an eight-core CPU.*

These relationships (with eight-core parallelization) can be seen in figure 3, which displays the following conclusions regarding their runtime relationships:

1. The fast multipole method has a much higher proportionality coefficient k on average than Brute-Force calculation and the Barnes-Hut approximation, making it far slower than both methods at low N (roughly $N << 10^4$).
2. Despite requiring the construction of an octree, the Barnes-Hut approximation maintains an incredibly low proportionality constant k (even less than that of brute force in some cases) due to the incredible efficiency of the Morton-sorted linear octree data structure.

3.3.1 Recommended Use-Cases for Each Computation Method

Thus, combined with the accuracy limitation of the Barnes-Hut algorithm discussed in section 11 and the inherent strengths and weaknesses of each method previously discussed throughout section 3, we can determine ideal use-cases for each derivative computation method.

First, the brute force computation is ideal for very low N ($N << 10^3$) that are highly volatile (i.e., with many near-field interactions) or require high accuracy over long time periods. Brute-force computation is the gold-standard for accuracy and reasonably fast for low N, and as such, this scenario has been the main focus of prior fish-dipole simulations [3, 4, 8].

Second, the Barnes-Hut approximation is ideal for fairly large N (roughly $10^2 << N << 10^8$) in non-volatile sys-

tems (i.e., with predominantly far-field interactions) and over smaller time scales, assuming the use of an approximation within $0.1 < \theta \leq 1.0$. Within this range of N and θ, the Barnes-Hut approximation is incredibly fast, but due to the added approximation error from Barnes-Hut, volatile systems will produce non-negligible error quickly and long timescales will cause this error to compound greatly.

Third, the fast multipole method is ideal for very large N (roughly $10^5 << N$) regardless of volatility and for any large N (roughly $10^3 << N$) for volatile systems due to its high accuracy and fastest scaling law $\mathcal{O}(N)$. The leading drawback of the fast multipole method is its proportionality coefficient k leading to a much higher runtime at low N, where a direct calculation through brute force would be much more efficient and equally (if not more) accurate.

4 Numerical Integration

As stated above, equations 1 and 2 form a system of coupled first-order ordinary differential equations, and as such, these can be integrated using convential first-order integration schemes like Runge-Kutta methods. As such, FINDS implements a variety of variable-order adaptive and non-adaptive methods of various order (as seen in table 2). The intent in using both adaptive and non-adaptive integrators is to provide end-users with a more exact, dynamic choice in computing each subsequent integration step based on the specifics of their initial system.

For example, highly volatile initial states likely require incredibly precise calculations (and thus, very small time steps), so users interested in observing the system at a much larger evaluation time-step can use an adaptive integrator which would separate the evaluation time step and the computation time-step such that the computation is done at very small time-scales δt_i 0 and the saved frames are those at the evaluation time-steps.

Alternatively, highly stable systems likely don't require sensitive integration, and as such, non-adaptive integrators are ideal as they keep the computation time step constant at the evaluation time step, so the only time-frames that are computed are the time-frames saved to disk.

Given a user-defined evaluation time step Δt and an end time t_{f0}, the goal of the integration module is to return the associated system states at the times $\{0, \Delta t, 2\Delta t, \ldots, T\Delta t\}$ where

$$T = \text{floor}(t_{f0}/\Delta t) \tag{28}$$

is the number of time-steps in the simulation.

To do so, the simulation integrates over the time domain of $t \in [0, T_f]$. Non-adaptive integrators will hold the computation time-step δt_i (where, in this context, i is the time-step index) constant as the evaluation time-step ($\delta t_i = \Delta t$), and

Method	Order	Adaptive?
Forward Euler	1	No
RK2	2	No
RK4	4	No
RK5	5	No
RK6	6	No
Bogacki-Shampine RK3(2)	3	Yes, 2nd Order
Dormand-Prince RK5(4)	5	Yes, 4th Order

Table 2: *List of the numerical integrators implemented in FINDS. If it is adaptive, the error computation order is also included.*

adaptive integrators will compute, for every time-state $\vec{\mathbf{X}}_i$ and δt_i, the advanced state $\vec{\mathbf{X}}_{i+1}$ and a new computation time-step δt_{i+1}, computed as

$$\delta t_{i+1} = \delta t_i \eta \left(\frac{\xi_i}{E_i} \right)^{\frac{1}{p}} \tag{29}$$

where $\eta = 0.9$ is a safety parameter and ξ_i is the tolerance, computed as

$$\xi_i = \gamma + \mu \max \left(\left\| \vec{\mathbf{X}}_{i+1} \right\|, \left\| \vec{\mathbf{X}}_i \right\| \right) \tag{30}$$

where γ and μ are the user-specified absolute and relative tolerance respectively, and E_i is the error of time-step i, computed as

$$E_i = \left\| \vec{\mathbf{X}}_{p,i+1} - \vec{\mathbf{X}}_{q,i+1} \right\| \tag{31}$$

where p is the computation order and q is the error order (for example, for RK5(4), $p = 5$ and $q = 4$) and $\vec{\mathbf{X}}_{p,i}$ refers to the p-th order solution for step i [17].

5 Database Management

The evaluation times $\{0, \Delta t, 2\Delta t, \ldots, T_f\}$ and their corresponding system states $\{\vec{\mathbf{X}}_1, \vec{\mathbf{X}}_2, \ldots \vec{\mathbf{X}}_{T_f}\}$ are written out to an HDF5 file using the format specified in table 3. HDF5 format was chosen over unstructured formats like raw binary due to its simplicity and portability and over common plaintext formats due to its efficiency and compactness [18].

This is achieved using FINDS' internal "datastream" object, which serves as a wrapper around the HDF5 API. The intent of this simplistic layout is for FINDS databases to be easily read using any valid HDF5 parser across any language after performing a simulation, and this is employed within FINDS to allow users to perform calculations quickly in C with FINDS, store their calculation results efficiently with HDF5, then open the data for analysis in C (using the datastream module), Python, and more (see section 8).

Path	Dimensions	Rank
/time	T	1
/volumetric_flow_rate	$T \times N$	2
/length	$T \times N$	2
/position	$T \times N \times 3$	3
/orientation	$T \times N \times 3$	3

Table 3: *The HDF5 layout for the FINDS databases. As defined above, T is the number of time-steps for a given simulation and N is the number of swimmers in the system of fish. The four bottom rows represent all of the unique data for an evolution trajectory of a system, or $\vec{\mathbf{X}}(t)$.*

6 Initial System Creation

The goal of FINDS is to evolve an initial fish system state $\vec{\mathbf{X}}_0$ to produce a trajectory, $\vec{\mathbf{X}}(t)$. Although the numerical calculation, data storage, and post-processing routines as described above are certainly important for studying school behavior, the precise creation of specific starting states $\vec{\mathbf{X}}_0$ is undoubtedly the most important task for producing effective simulations. As such, FINDS offers a small but powerful suite of options for producing various (mostly symmetric) initial systems.

To begin, the most precise construction of a FINDS system must be done directly. FINDS system objects are fundamentally just variable-size lists of vectors, so end users desiring systems that are highly asymmetrical or specific by directly writing the data describing each swimmer, i.e., $\vec{\mathbf{x}}_{c,i}$, $\hat{\mathbf{n}}_i$, σ_i, and ℓ_i for every swimmer i.

For systems that are slightly more symmetric or general (such as requiring simple 3-D geometries like cubes, balls, or spheres), FINDS contains a streamlined process for generating them through initial conditions, transformations, and combinations.

6.1 Initial Conditions

The initial conditions that describe a system are best understood under three main categories: the distribution options, which determine $\vec{\mathbf{x}}_{c,i}$, the orientation options, which determine $\hat{\mathbf{n}}_i$, and the constant options, which determine σ_i and ℓ_i.

Next, the orientation options determines how the orientation of each fish is determined, and this is generated after the distribution as many of the orientations are dependent on the position of each swimmer. After generating the starting orientations based on the position of the swimmer as outlined in table 5, orientations can also be randomly perturbed by some maximum angle ϕ defined by the user to introduce some random instability in the system. This perturbation is done by randomly selecting a roll $\phi_x \in [-\phi, \phi]$, pitch $\phi_y \in [-\phi, \phi]$, and yaw $\phi_z \in [-\phi, \phi]$, then rotates the orientations using the 3-dimensional rotation matrix

Type	Parameters	Description
Cube	Side Length, Spacing	Cubic Lattice
Sphere	Radius, Spacing	Fibonacci Sphere
Ball	Radius, Spacing	Ball Lattice
Random	N, Absolute Bound	Random Distribution

Table 4: *The distribution options in FINDS and their associated parameters. Note that the "random" distribution produces N random points within a cube centered at the origin that goes from $-L$ to L on every dimension where L is the absolute bound.*

$$\mathbf{R}(\phi_x, \phi_y, \phi_z) = \mathbf{R}_z(\phi_z)\mathbf{R}_y(\phi_y)\mathbf{R}_x(\phi_x) \tag{32}$$

where $\mathbf{R}_i(\theta)$ is the associated rotation matrix for a a rotation around axis i by angle θ.

Name	Equation
Radial Inward	$-\vec{\mathbf{r}}_i$
Radial Outward	$\vec{\mathbf{r}}_i$
Swirl Inward	$\frac{-(\sin\theta_i+\cos\theta_i)\hat{\mathbf{x}}+(\cos\theta_i-\sin\theta_i)\hat{\mathbf{y}}}{\sqrt{2}}$
Swirl Outward	$-\sin\theta_i\hat{\mathbf{x}} + \cos\theta_i\hat{\mathbf{y}}$
Saddle	$-\sin\theta_i\hat{\mathbf{x}} - \cos\theta_i\hat{\mathbf{y}}$
Aligned	$\hat{\mathbf{x}}$

Table 5: *The orientation options. Each equation describes the un-normalized orientation of each swimmer, which is later normalized $\theta_i = \text{atan}(y_i/x_i)$ and $\vec{\mathbf{r}}_i$ is the position of swimmer i, $\vec{\mathbf{r}}_i = \langle x_i, y_i, z_i \rangle$.*

Lastly, there are the constant generation options for the length and volumetric flow rate for each swimmer. This is essentially just whether these constants are set as one uniform value for all swimmers. The user has the ability to select a minimum and maximum value for each or a singular value to apply to all of the swimmers.

6.2 Transformations and Combinations

To create more interesting systems (such as witnessing the interaction between groups of pre-defined system geometries), users can perform transformations and combinations on systems produced using the aforementioned initial conditions. A simple analogy to this is like thinking of using the above initial conditions to produce individual schools of fish, and the overall system $\vec{\mathbf{X}}$ describes all of the swimmers in the simulation (so potentially multiple schools).

Two transformations are available in FINDS for systems: rotation and translation. Given a system, users can rotate a system given a roll ϕ_x, pitch ϕ_y, and yaw ϕ_z using the 3-dimensional rotation matrix $\mathbf{R}(\phi_x, \phi_y, \phi_z)$ of equation 32,

and users can translate a system using a translation vector $\Delta\vec{\mathbf{r}} = \langle \Delta x, \Delta y, \Delta z \rangle$.

With these transformations, fish systems can also be copied and combined to produce greater systems. For example, a system of two colliding ball-shaped schools was produced in the "collision" example (see section 10). To set up this initial system: first, a ball of an arbitrary radius r and spacing is produced with an aligned orientation, and this can be denoted $\vec{\mathbf{X}}_a$. Then, this is copied to produce a second school $\vec{\mathbf{X}}_b$, and then $\vec{\mathbf{X}}_a$ is translated by $1.5r\hat{\mathbf{x}}$ and $\vec{\mathbf{X}}_b$ is translated by $-1.5r\hat{\mathbf{x}}$. Lastly, $\vec{\mathbf{X}}_b$ is rotated by $\phi_z = \pi$, and the two systems are combined into one overall initial system $\vec{\mathbf{X}}_0$, and this is what is used for computations.

7 Simulation Configuration Files

For simple simulations (i.e., those that do not require specific, complex starting systems as would need to be created manually as described under section 6), the `simulate` program can be used for performing simulations from TOML configuration files (which are processed using the TOMLC17 library) [19, 20]. However, these files don't allow for the creation of complex compound systems using any transformations or combinations. These files simplify the process of performing simulations and storing or sharing the initial conditions of a simulation. These files are split into four sections: "system," which describes the initial system generation options as described above, "differentiation," which describes the derivative computation method, "integration," which describes the integration method, and "debug," which describes the debug printing options (i.e., logging).

8 Post-Processing in Python

After running a simulation with FINDS, the simulation will output a data file under `output/simulations/sim-.../data.h5` (where `...` is the date and time of when the simulation was run). As stated in section 5, users can directly read their HDF5 database in the language of their choosing if they so wish, but alternatively, to simplify the post-processing of simulation data files, FINDS includes a simple framework for performing post-processing data analysis in Python. This is done using the `process_data.py` script. This script works as the central module for post-processing by controlling and executing "Processing Modules," or objects defined with an initialization function to executed before reading the dataset, an update function to be executed on each frame of the dataset, and a closing function to be executed after reading the dataset. For example, the `AnimationGenerator` module uses its initialization function for initializing the MatPlotLib animation, its update function for drawing a frame to the animation, and its closing function for saving the figure. The full list of these modules can be seen in table 6.

Name	Purpose
Animation Generator	Generates an MP3 animation of the system evolution.
Cross Section Generator	Plots the $xy, z = 0$ plane at $t = 0$ if any swimmers exist on it to PNG or PDF.
Density Animation Generator	Generates an MP3 animation of the radial density distribution from the center-of-mass over time.
Mean Radial Distance Generator	Plots the mean and standard deviation in the radial distance from the center-of-mass as a function of time to PNG or PDF.
Snapshot Generator	Plots the first, middle, and last frames of the simulation (from AnimationGenerator) to PNG or PDF.
Trajectory Plotter	Plots the trajectories of each swimmer in the system in 3-D space to PNG or PDF.

Table 6: *A list of the default Python post-processing modules available in FINDS.*

9 Validation

FINDS' derivative calculation and integration methods were validated for correctness through both unit testing and comparison with prior implementations of the source/sink dipole model for swimmers. Unit tests were primarily used for simple, easily-testable functions like data I/O, and the overall simulation behavior was validated visually and numerically against simulations produced using previous Python and MATLAB implementations of the dipole model.

In particular, FINDS was validated in two dimensions by reproducing figure 8 of Mabrouk and Floryan 2025, which contains four scenarios of the trajectory evolution for two coplanar ($z = 0$) swimmers based on the distances from their centers of mass on the x and y axes and the angle between their orientations $\Delta\theta$, and this can be observed in figure 4 [4]. Visually, these plots match, and as such, numerical validation (by directly computing the difference between the expected calculation and the observed calculation from FINDS) alongside validation for a greater N and three-dimensional system is needed, and this is done using the validation observed in figure 5. This numerical computation is produced by calcu-

lating an error score between the expected state $\vec{\mathbf{X}}_t$ and the produced state $\vec{\mathbf{X}}'_t$ for each time t as

$$E_t = \sqrt{\sum_{i=1}^{N} \left(\left\| \vec{\mathbf{x}}_{c,i} - \vec{\mathbf{x}}'_{c,i} \right\|^2 + \left\| \vec{\mathbf{n}}_i - \vec{\mathbf{n}}'_i \right\|^2 \right)} \tag{33}$$

where $\vec{\mathbf{x}}_{c,i}$ and $\vec{\mathbf{x}}'_{c,i}$ are the expected and computed center of mass position and $\vec{\mathbf{n}}_i$ and $\vec{\mathbf{n}}'_i$ are the expected and computed orientation for each swimmer i at time t.

As can be seen in figure 5, the error for the three coplanar cases is entirely zero, showing that for two-dimensional, two-swimmer simulations, the results of FINDS are entirely identical to those of previous implementations. However, in the 13-swimmer, three-dimensional case, only the brute-force and fast multipole method calculations produce negligible error, and the Barnes-Hut approximation produces non-negligible error that compounds over the course of the simulation due to the inherent approximation error in using a weighted average dipole (see section 11).

10 Examples

FINDS comes with three basic examples of performing simulations directly using the C API: `collision.c`, which collides two schools of fish as described in section 6.1, `imploding_sphere.c`, which produces a Fibonacci sphere with a radial inward orientation which simulates an implosion, and `rotating_sphere.c`, which produces a ball that swirls outward and expands. To demonstrate the usage of FINDS in detail, we will describe, in full, the example of the imploding sphere.

First, the initial state $\vec{\mathbf{X}}_0$ is generated with a radius r and a spacing $r/2$. Each swimmer was given a uniform length and volumetric flow rate $\ell = 1$ and $\sigma = 4\pi$ such that the self-propelled speed for each swimmer becomes $U = 1$. This initial generated system can be seen in figure 6.

Then, the derivative computation method is set as the Barnes-Hut approximation with $\theta = 1$ using regularization at $\varepsilon = 10^{-6}$, and the integration computation method is set as non-adaptive fourth-order Runge-Kutta with an evaluation time-step of $\Delta t = 1$ seconds and an end time of $T_f = 100$ seconds.

After running the simulation and processing the produced dataset using the modules described in section 8, the mean radial distance plot reflects the movement of swimmers towards the origin before being violently ejected, and this can be observed in figure 7.

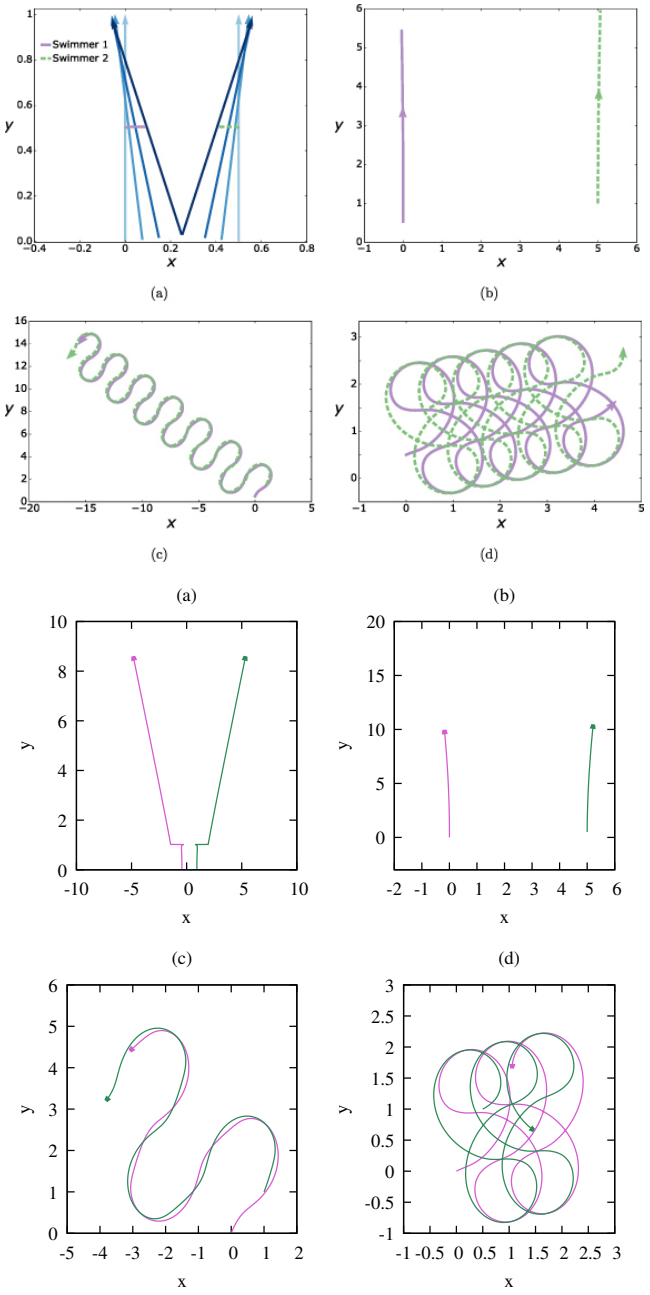

Figure 4: *The coplanar, two-swimmer trajectory plots shown in figure 8 of Mabrouk and Floryan 2025 (top) and its reproduction with FINDS (bottom). Note that the two simulations need not exactly match in time as the observed behavior in each case is stable, thus only a few repetitions are necessary to observe similar behavior. Additionally, figure (a) matches the correct behavior, as the original figure was simply conveying the growing separation between the two swimmers over time.*

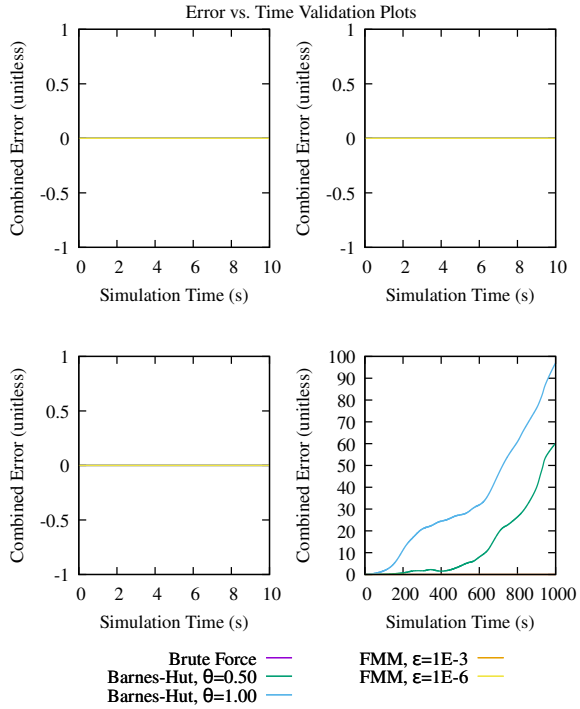

Figure 5: *The numerical validation comparison plots of the error in the simulation versus the time. The top left, top right, and bottom left plots are two-swimmer, coplanar simulations. The bottom right plot is a 13-swimmer trajectory from Figure 9(b) of Mabrouk and Floryan 2025 [4].*

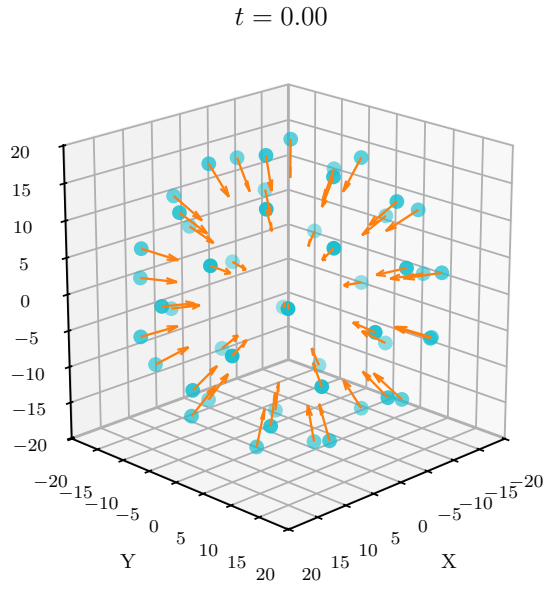

Figure 6: *The initial system generated for the implosion example.*

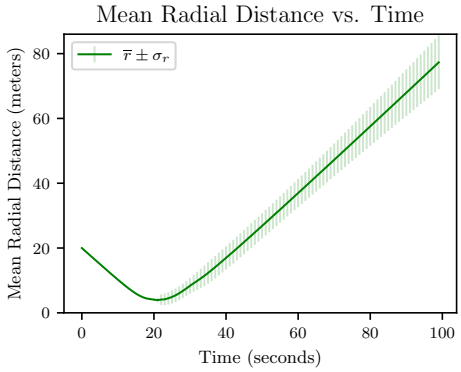

Figure 7: *The mean and standard deviation of radial distance as a function of time for the implosion example.*

11 Limitations

FINDS has several critical limitations, and it is key to understand these limitations when performing simulations. The first key limitation lies in our implementation of the Barnes-Hut approximation as described in section 3.1. The Barnes-Hut approximation performs averages at the dipole level despite interactions being computed at the monopole level, so this approximation introduces a small amount of error (assuming short fish lengths and far-field interactions only) that compounds considerably over time (as was observed in figure 5). A potential solution to this problem would be to switch to a higher-order (dipole) implementation of the Barnes-Hut approximation such as that of Marchevsky et al. 2023 [21], but we have chosen to forego this as the main cases where the expansion's error is non-negligible is in close-field interactions (where the far-field model is already incorrect unless heavily regularized) and for very large fish lengths (which is patently unrealistic). Additionally, more importantly, the fundamental solution to this issue is to simply use a higher-order multipole expansion to perform the approximation, and this can simply be done by using the fast multipole method instead of the Barnes-Hut approximation (thus why the fast multipole method produces much less error than the Barnes-Hut approximation in figure 5).

Additionally, each Morton number is stored as a 64-bit unsigned integer. These numbers are intended to simultaneously encode the x, y, and z positions as unsigned integers, thus each dimension must be able to be stored in only 21 bits. As the maximum unsigned integer that can be represented in 21 bits is $2^{21} - 1$ or approximately 2.1 million. As a consequence, two fish whose centers-of-mass are equal in integral values (i.e., when the float precision is ignored) are considered to be at exactly the same location during Octree construction, and the maximum distance that any fish can be in FINDS (without losing data or corrupting the simulation)

is $2^{21} - 1$ *on each axis*. However, these two issues are largely negligible for simulations in FINDS. First, FINDS is intended for far-field interactions, so two fish having the exact same integral position is incredibly unlikely unless this was directly intended by the user, and even in such a case, any two fish with equal Morton numbers simply have a random insertion order with little to no effect on the velocity calculation. Second, it is highly unlikely for fish positions to evolve past the aforementioned limit over the course of most simulations as far-field interactions produce typically small velocity fields.

12 Future Work

The most pertinent future addition to FINDS would be the creation of bindings in other languages such as Python or Julia such that users can generate simulations and perform simulations in a much more streamlined and flexible way. The next major change would be implementing in a path-independence that no longer requires users to clone the FINDS repository and develop their simulations only within this directory. Transitioning FINDS to a path-independent system that links as a library located on the system-level (such as a PIP module for the Python API or a C library linked globally) that reads and writes to either the same directory where the script invoking FINDS is run from or a separate directory defined by the user would allow simulations to be localized to simple scripts or Jupyter notebooks while still taking advantage of the immense speed of the underlying C code.

Another major potential inclusion into FINDS would be the implementation of a vortex-based dipole model (like that of Tchieu et al. 2012) rather than a Coulombic-based dipole model to widen the range of simulation options for users as-needed and to allow for the rigorous comparison of the two models at large N [8].

Additionally, small alternative improvements could be made in various places throughout the FINDS codebase. An option to utilize a normal distribution centered around the middle of a specified range for randomly-selected constants ℓ and σ (as described in section 6.1) instead of the default uniform distribution may be a preferable option for some users due to being considerably more realistic. Furthermore, as stated in section 11, a dipole-order implementation of the Barnes-Hut approximation could be implemented.

13 Conclusion

Altogether, we present FINDS, a simple yet powerful N-body simulator for simulating the evolution of schools of fish due to passive hydrodynamics. This software enables the simulation of massive group sizes, potentially millions or hundreds of millions, by implementing two standard N-body interaction algorithms: the Barnes-Hut approximation and the fast multipole method, and these methods reduce the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ or $\mathcal{O}(N)$. Furthermore, FINDS is a complete, modular, and simple software suite for performing and analyzing simulations, containing several miniature libraries for defining the initial conditions in C and analyzing the produced data in Python. A simple utility for performing simulations using pre-defined simulation files is also included. Given that the benchmark test proves that FINDS produces the increase in efficiency that was intended through implementing the Barnes-Hut and fast multipole method algorithms and that the validation test proves that FINDS is accurate and precise within the range of expected error, we introduce FINDS as a capable and robust simulation system to aid research into massive schools of fish.

14 Acknowledgements

In the development of FINDS, Alvin (Ching-Yin) Ng's `grav_sim` was used extensively as a reference for implementing a fast N-body simulator in C, particularly in implementing of the Barnes-Hut algorithm [22]. In addition, the direct advice of Alvin Ng was instrumental to the implementation of linear octrees and CPU parallelization with OpenMP.

Furthermore, generative artificial intelligence (Anthropic's Sonnet 5 Claude, OpenAI's ChatGPT-5.5 and Google's Gemini 3.3) were used heavily for initial code generation before being thoroughly tested and validated to ensure correctness. Generative artificial intelligence was not used in developing the content of this report whatsoever, including the abstract, section text, figures, or captions, but artificial intelligence tools "checkmymanuscript" and Grammarly were used for proofreading spelling and grammar mistakes.

Code Availability

FINDS can be presently obtained under the `mufaro3/finds` GitHub repository, and the release build used for this paper is permanently archived on Zenodo [23].

References

[1] Julia K. Parrish, William M. Hamner, and Charles T. Prewitt. "Introduction – From individuals to aggregations: Unifying properties, global framework, and the holy grails of congregation". In: *Animal Groups*

in Three Dimensions: How Species Aggregate. Ed. by Julia K. Parrish and William M.Editors Hamner. Cambridge University Press, 1997, pp. 1–14.

[2] Tamás Vicsek and Anna Zafeiris. "Collective motion". In: *Physics Reports* 517.3 (2012). Collective motion, pp. 71–140. ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2012.03.004. URL: https://www.sciencedirect.com/science/article/pii/S0370157312000968.

[3] Mohamed Niged Mabrouk and Daniel Floryan. "Group cohesion and passive dynamics of a pair of inertial swimmers with three-dimensional hydrodynamic interactions". In: *Bioinspiration and Biomimetics* 20.1 (Nov. 2024), p. 016014. DOI: 10.1088/1748-3190/ad936d. URL: https://doi.org/10.1088/1748-3190/ad936d.

[4] Mohamed Niged Mabrouk and Daniel Floryan. "Effects of symmetry and hydrodynamics on the cohesion of groups of swimmers". In: *Bioinspiration and Biomimetics* 20.6 (Aug. 2025), p. 066005. DOI: 10.1088/1748-3190/ae0bd9. URL: https://doi.org/10.1088/1748-3190/ae0bd9.

[5] Marco Dorigo, Guy Theraulaz, and Vito Trianni. "Reflections on the future of swarm robotics". In: *Science Robotics* 5.49 (2020), eabe4385. DOI: 10.1126/scirobotics.abe4385. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.abe4385. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abe4385.

[6] Craig W. Reynolds. "Flocks, herds and schools: A distributed behavioral model". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: Association for Computing Machinery, 1987, pp. 25–34. ISBN: 0897912276. DOI: 10.1145/37401.37406. URL: https://doi.org/10.1145/37401.37406.

[7] Iain D. Couzin et al. "Effective leadership and decision-making in animal groups on the move". In: *Nature* 433.7025 (Feb. 2005), pp. 513–516. ISSN: 1476-4687. DOI: 10.1038/nature03236. URL: https://doi.org/10.1038/nature03236.

[8] Andrew A. Tchieu, Eva Kanso, and Paul K. Newton. "The finite-dipole dynamical system". In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 468.2146 (May 2012), pp. 3006–3026. ISSN: 1364-5021. DOI: 10.1098/rspa.2012.0119. eprint: https://royalsocietypublishing.org/rspa/article-pdf/468/2146/3006/821067/rspa.2012.0119.pdf. URL: https://doi.org/10.1098/rspa.2012.0119.

[9] Ed Yong. *What makes 250,000,000 fish gather in the same place?* Mar. 2009. URL: https://www.nationalgeographic.com/science/article/what-makes-250000000-fish-gather-in-the-same-place.

[10] Ananth Y. Grama et al. "N-Body Computational Methods". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1259–1268. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_97. URL: https://doi.org/10.1007/978-0-387-09766-4_97.

[11] Josh Barnes and Piet Hut. "A Hierarchical O(N log N) Force-Calculation Algorithm". In: *Nature* 324 (1986), pp. 446–449. DOI: 10.1038/324446a0. URL: https://www.nature.com/articles/324446a0.

[12] Leslie Greengard and Vladimir Rokhlin. "A fast algorithm for particle simulations". In: *Journal of Computational Physics* 73.2 (1987), pp. 325–348. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(87)90140-9. URL: https://www.sciencedirect.com/science/article/pii/0021999187901409.

[13] Travis Askham et al. *FMM3D*. Feb. 2026. URL: https://fmm3d.readthedocs.io.

[14] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 6.0*. Nov. 2024. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf.

[15] Zuo-Ming Yin et al. "A New Optimal Parallel Algorithm for Generating Linear Level Octree from Voxels". In: *Singapore Supercomputing Conference '90: Supercomputing for Strategic Advantage*. Ed. by Kang Hoh Phua and Kia Fock Loe. World Scientific, 1991, pp. 164–181.

[16] Pablo D. Viñambres et al. *Efficient Neighbourhood Search in 3D Point Clouds Through Space-Filling Curves and Linear Octrees*. 2026. arXiv: 2603.06771 [cs.CG]. URL: https://arxiv.org/abs/2603.06771.

[17] Joakim Sundnes. "Adaptive Time Step Methods". In: *Solving Ordinary Differential Equations in Python*. Cham: Springer Nature Switzerland, 2024, pp. 61–77. ISBN: 978-3-031-46768-4. DOI: 10.1007/978-3-031-46768-4_4. URL: https://doi.org/10.1007/978-3-031-46768-4_4.

[18] The HDF Group. *Hierarchical Data Format, version 5*. Version 2.0.0. Dec. 2025. DOI: 10.5281/zenodo.17808614. URL: https://doi.org/10.5281/zenodo.17808614.

[19] Tom Preston-Werner. *TOML: Tom's Obvious Minimal Language*. Version 1.1.0. 2026. URL: https://toml.io/en/.

[20] Cheng Tan. *TOMLC17: A lightweight, strictly compliant TOML v1.1 parser for C and C++*. Version 1.1-R260618. June 2026. URL: https://github.com/cktan/tomlc17.

[21] Ilia Marchevsky, Evgeniya Ryatina, and Alexandra Kolganova. "Fast Barnes–Hut-based algorithm in 2D vortex method of computational hydrodynamics". In: *Computers and Fluids* 266 (2023), p. 106018. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2023.106018. URL: https://www.sciencedirect.com/science/article/pii/S0045793023002438.

[22] Ching-Yin Ng. *grav_sim: N-body gravity simulation library with C and Python API*. Version 1.0.0. May 2025. URL: https://alvinng4.github.io/grav_sim/.

[23] Mufaro J.M. *mufaro3/finds: Zenodo release*. Version zenodo. Aug. 2026. DOI: 10.5281/zenodo.21970758. URL: https://doi.org/10.5281/zenodo.21970758.